Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ

по лабораторной работе №4

на тему

БАЗОВЫЕ АЛГОРИТМЫ НА GPU

Вариант 1

Студенты: Е.А. Петрович

М.А. Ходосевич

Преподаватель: Т.С. Жук

МИНСК 2024

**1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**1.1 Цель работы**

Изучить особенности разработки алгоритмов на GPU.

**1.2 Исходные данные к работе**

Реализовать один из следующих алгоритмов (в соответствии с заданным вариантом):

1. Интегральная сумма одномерного массива (в пределах блока потоков (бонус, если кто-то решит задачу в пределах всей матрицы).

2. Редукция массива (меняется операция, которая выполняется, - плюс, минус, xor, or, and, …).

3. Сортировка заданного массива.

Задача - придумать и объяснить (описать) алгоритм.

Обязательные требования: идентичны обязательным требованиям к лабораторной работе № 3 (реализация CPU и GPU, сравнение времени в едином формате, вывод фрагмента массива на экран, инициализация массива случайными числами). Опциональные условия: идентичны опциональным требованиям к лабораторной работе № 3.

Дополнительно можно реализовать ещё один алгоритм:

1. Расчёт гистограммы.

Сравнить результаты работы реализованных алгоритмов при прочих равных.

**2 ВЫПОЛНЕНИЕ РАБОТЫ**

В лабораторной работе необходимо реализовать алгоритм интегральной суммы одномерного массива. Интегральная сумма – алгоритм, применяющийся в интегральных изображениях – алгоритм, позволяющий эффективно вычислять сумму значений, заключенных в прямоугольном подмножестве многомерного массива. Алгоритм работы приведен на рисунке 2.1.

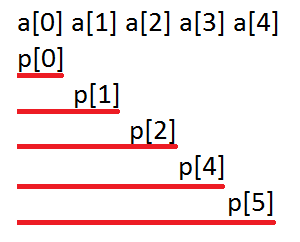


Рисунок 2.1 – Алгоритм интегральной суммы

На рисунке видно, что алгоритм работает по принципу: каждый последующий элемент равен сумме всех предыдущих (первый элемент выходного массива равен первому элементу входного, второй элемент выходного массива равен сумме первого и второго элементов входного массива и так далее).

Для программы на CPU был использован обычный цикл, который проходит по всем элементам массива. Для GPU используются две функции – первая вычисляет интегральную сумму в пределах блока, вторая вычисляет интегральную сумму в пределах всего массива, объединяя блоки между собой.

Результаты работы процессоров при разной длине массива приведены в таблице 2.1.

Таблица 2.1 – Результаты работы

|  |  |  |  |
| --- | --- | --- | --- |
| Длина | CPU, микросекунд | GPU, микросекунд | GPU/CPU |
| 100000 | 1867 | 1286 | 1,45 |
| 1000000 | 7809 | 4333 | 1,80 |
| 5000000 | 40157 | 17765 | 2,26 |
| 10000000 | 87965 | 33865 | 2,59 |

**3 ВЫВОДЫ**

В ходе лабораторной работы были изучены особенности разработки алгоритмов на GPU. В соответствии с выданным вариантом был реализован алгоритм интегрального суммирования одномерного массива с типом данных int. Были выполнены все требования к лабораторной, приведен алгоритм и его объяснение. Были представлены итоговые результаты работы. Так как все работает и отображается корректно, можно сделать вывод, что лабораторная работа проведена успешно.

**ПРИЛОЖЕНИЕ А**

*(обязательное)*

Исходный текст программы

Содержимое файла kernel.cu:

1. #include <iostream>
2. #include <cuda\_runtime.h>
3. #include <chrono>
4. #include <vector>
5. #include <algorithm>
6. constexpr int LENGTH = 1000000;
7. constexpr int MIN\_VALUE = 1;
8. constexpr int MAX\_VALUE = 50;
9. void checkCudaError(cudaError\_t err, const char\* msg) {
10. if (err != cudaSuccess) {
11. std::cerr << "CUDA Error (" << msg << "): " << cudaGetErrorString(err) << std::endl;
12. exit(EXIT\_FAILURE);
13. }
14. }
15. std::vector<int16\_t> prefix\_sum\_cpu(const std::vector<int16\_t>& input\_array) {
16. int input\_length = input\_array.size();
17. std::vector<int16\_t> result(input\_length, 0);
18. auto start\_time = std::chrono::high\_resolution\_clock::now();
19. result[0] = input\_array[0];
20. for (int i = 1; i < input\_length; ++i) {
    1. result[i] = result[i - 1] + input\_array[i];
21. }
22. auto end\_time = std::chrono::high\_resolution\_clock::now();
23. auto exec\_time = std::chrono::duration\_cast<std::chrono::microseconds>(end\_time - start\_time);
24. std::cout << "Time CPU: " << exec\_time.count() << " microseconds" << std::endl;
25. return result;
26. }
27. \_\_global\_\_ void prefix\_sum\_kernel(const int16\_t\* input, int16\_t\* result, int\* block\_sums, int length) {
28. extern \_\_shared\_\_ int16\_t temp[];
29. int tid = threadIdx.x;
30. int idx = threadIdx.x + blockIdx.x \* blockDim.x;
31. if (idx < length) {
    1. temp[tid] = input[idx];
32. }
33. else {
    1. temp[tid] = 0;
34. }
35. \_\_syncthreads();
36. for (int offset = 1; offset < blockDim.x; offset \*= 2) {
    1. int val = 0;
    2. if (tid >= offset) {
       1. val = temp[tid - offset];
    3. }
    4. \_\_syncthreads();
    5. temp[tid] += val;
    6. \_\_syncthreads();
37. }
38. if (idx < length) {
    1. result[idx] = temp[tid];
39. }
40. if (tid == blockDim.x - 1 && block\_sums != nullptr) {
    1. block\_sums[blockIdx.x] = result[idx];
41. }
42. }
43. \_\_global\_\_ void add\_block\_sums(int16\_t\* result, int\* block\_sums, int length) {
44. int idx = blockIdx.x \* blockDim.x + threadIdx.x;
45. if (blockIdx.x > 0 && idx < length) {
    1. int correction = 0;
    2. for (int i = 0; i < blockIdx.x; i++) {
       1. correction += block\_sums[i];
    3. }
    4. result[idx] += correction;
46. }
47. }
48. std::vector<int16\_t> prefix\_sum\_gpu(const std::vector<int16\_t>& input\_array) {
49. int input\_length = input\_array.size();
50. std::vector<int16\_t> result(input\_length, 0);
51. int16\_t\* d\_input = nullptr;
52. int16\_t\* d\_result = nullptr;
53. int\* d\_block\_sums = nullptr;
54. checkCudaError(cudaMalloc(&d\_input, input\_length \* sizeof(int16\_t)), "cudaMalloc (d\_input)");
55. checkCudaError(cudaMalloc(&d\_result, input\_length \* sizeof(int16\_t)), "cudaMalloc (d\_result)");
56. checkCudaError(cudaMalloc(&d\_block\_sums, (input\_length / 32 + 1) \* sizeof(int)), "cudaMalloc (d\_block\_sums)");
57. checkCudaError(cudaMemcpy(d\_input, input\_array.data(), input\_length \* sizeof(int16\_t), cudaMemcpyHostToDevice), "cudaMemcpy HostToDevice");
58. int threads\_per\_block = 32;
59. int blocks = (input\_length + threads\_per\_block - 1) / threads\_per\_block;
60. cudaEvent\_t start, stop;
61. cudaEventCreate(&start);
62. cudaEventCreate(&stop);
63. cudaEventRecord(start);
64. prefix\_sum\_kernel << <blocks, threads\_per\_block, threads\_per\_block \* sizeof(int16\_t) >> > (d\_input, d\_result, d\_block\_sums, input\_length);
65. cudaEventRecord(stop);
66. cudaEventSynchronize(stop);
67. add\_block\_sums << <blocks, threads\_per\_block >> > (d\_result, d\_block\_sums, input\_length);
68. float milliseconds = 0;
69. cudaEventElapsedTime(&milliseconds, start, stop);
70. std::cout << "Time GPU: " << milliseconds \* 1000 << " microseconds" << std::endl;
71. checkCudaError(cudaMemcpy(result.data(), d\_result, input\_length \* sizeof(int16\_t), cudaMemcpyDeviceToHost), "cudaMemcpy DeviceToHost");
72. cudaFree(d\_input);
73. cudaFree(d\_result);
74. cudaFree(d\_block\_sums);
75. cudaEventDestroy(start);
76. cudaEventDestroy(stop);
77. return result;
78. }
79. void print\_array(const std::vector<int16\_t>& array, const std::string& label, int limit = 10) {
80. std::cout << label << ": ";
81. for (size\_t i = 0; i < limit && i < array.size(); ++i) {
    1. std::cout << array[i] << " ";
82. }
83. std::cout << std::endl;
84. }
85. int main() {
86. std::vector<int16\_t> input\_array(LENGTH);
87. std::generate(input\_array.begin(), input\_array.end(), []() {
    1. return rand() % MAX\_VALUE + MIN\_VALUE;
    2. });
88. print\_array(input\_array, "Input Array");
89. std::vector<int16\_t> result\_cpu = prefix\_sum\_cpu(input\_array);
90. std::vector<int16\_t> result\_gpu = prefix\_sum\_gpu(input\_array);
91. print\_array(result\_cpu, "CPU Results");
92. print\_array(result\_gpu, "GPU Results");
93. return 0;
94. }